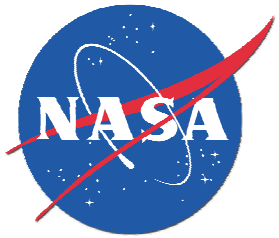
Code Execution and Runtime Verification

Jeff Zemerick



Outline

- **The source code.**
- The profiler.
- Executing unit tests.
- Runtime Verification.

Overview of the Code

- ~1.2 million SLOC
- Organized by functional module (~150 modules)
- The code in each directory is independent of other code (can be built separately).
- Compiles and executes on x86 Linux.
- Built as shared libraries but must be built statically.

Build Changes

- Components are compiled as shared libraries.
- Shared libraries cannot be easily instrumented.
- Modified the build process to do the build so that the executable is linked statically.
  - Determined what source files are needed (cross-module), build and link them with the unit tests.

Outline

- The source code.
- **The profiler.**
- Executing unit tests.
- Runtime Verification.

The Profiler

- Created by my 4 NEAP interns this summer.
- C profiler.
- Event-based (function entrances/exits).
- Captures execution trace and can export the trace as: plain text, CSV, XML
- Translates function addresses to function names.
- The interns did a fantastic job.

Instrumenting the Code

- Modified the makefile to include support for:
  - Profiling (add my interns' profiler object file when linking)
  - Debugging – allows for translation of (useless) function addresses to (useful) function names.

Example Execution Trace

```
main  (1)  (??)

|=-function1 (1)  (main)

|=-|=-function2  (1)  (function1)

|=-|=-function3  (1)  (function1)

|=-|=-|=-function4 (1)  (function3 )

|=-|=-|=-|=-function5(1)  (function4 )
```

Indentation shows depth.

Outline

- The source code.
- The profiler.
- **Executing unit tests.**
- Runtime Verification.

Executing the Unit Tests

- A wrapper facilitates the execution of the unit tests.
- The wrapper provides stubs for hardware-specific functionality.
  - Allows for testing the code on X86 Linux by providing stub functions for the hardware-specific functionality.

UTH Example

Event report generation for FSW build:

Event report generation for Test build:

Outline

- The source code.
- The profiler.
- Executing unit tests.
- **Runtime Verification.**

Runtime Verification

- Requirements for runtime verification:
  - Code that will compile and execute.
  - Ability to instrument the code to monitor the execution.
  - Ability to compare the execution with a model of the desired behavior.
- None of the FSW or unit tests were modified for this work.

Purpose

- Using the execution trace of the code, can we identify the presence of implemented requirements?

Why We Can Attempt to Answer This

- Unit tests achieve 100% coverage of module testing, per developer rule.
  - If a requirement has been implemented, it should be in the execution trace.

Untested lines include `default` statements in `switch` statements and code which is tested by other modules.

Filename	Coverage		
[redacted].c		99.2 %	968 / 976 lines
[redacted].c		17.6 %	248 / 1411 lines
[redacted].c		82.7 %	139 / 168 lines
[redacted].c		68.4 %	13 / 19 lines
[redacted].c		53.8 %	7 / 13 lines
[redacted].c		92.5 %	1245 / 1346 lines
[redacted].c		97.9 %	92 / 94 lines
[redacted].c		66.7 %	6 / 9 lines
[redacted].c		98.4 %	482 / 490 lines
[redacted].c		100.0 %	415 / 415 lines

Modeling the Behavior

- Model can be created in two forms:
  - Plain text
  - UML activity diagram (work in progress).
- Only one model per requirement is necessary.
- Which model type to create and use is up to the analyst.
- The behavior can be **desired** behavior or **undesired** behavior.

Plain Text Model for Event Reporting

Model Rules:

command: <command>

success: <result>

failure: < result >

At least one command.

Either Success, Failure, or both.

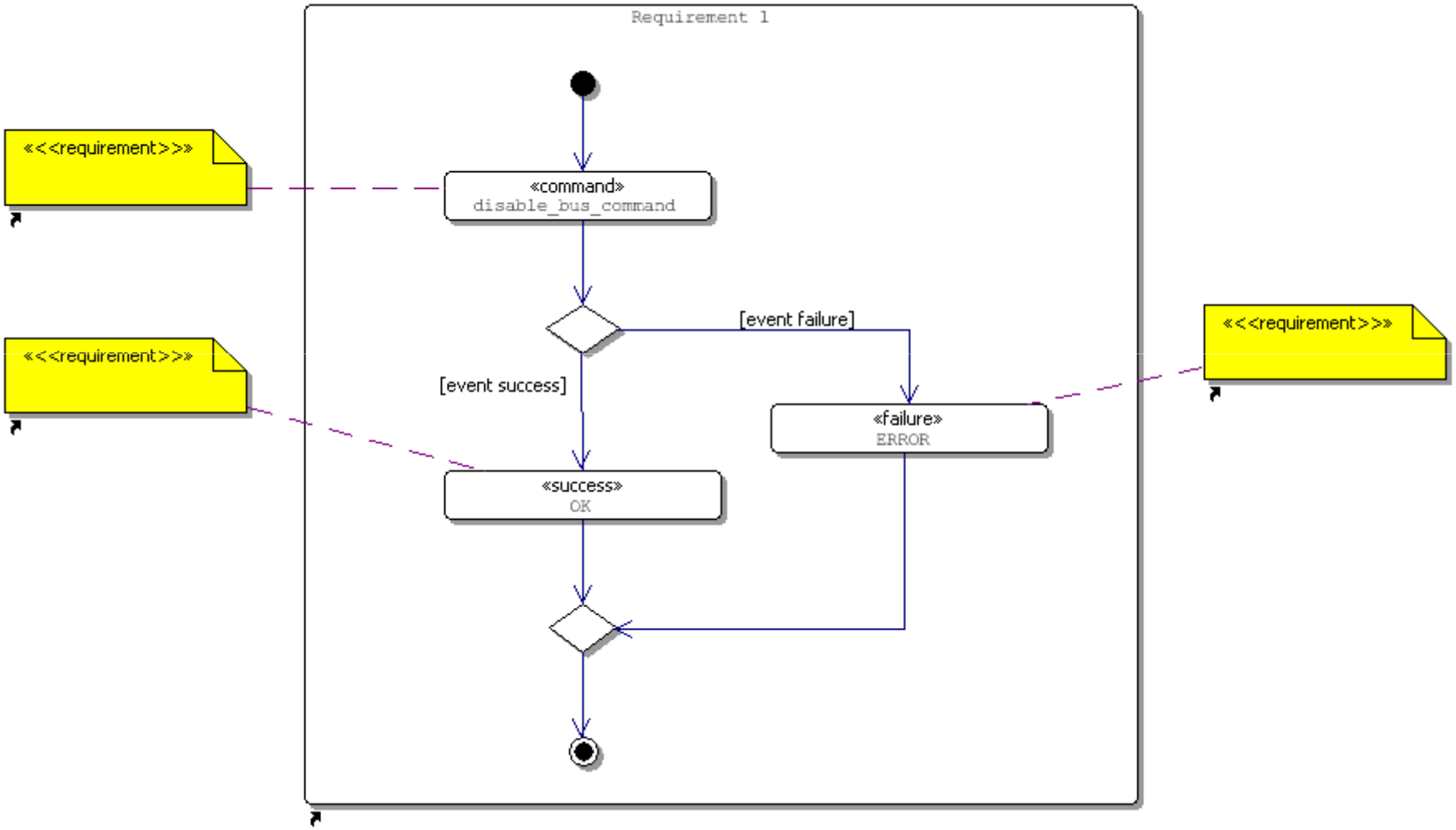
Example Model:

command: disable_bus_cmd

success: OK

failure: ERROR

Corresponding UML Model

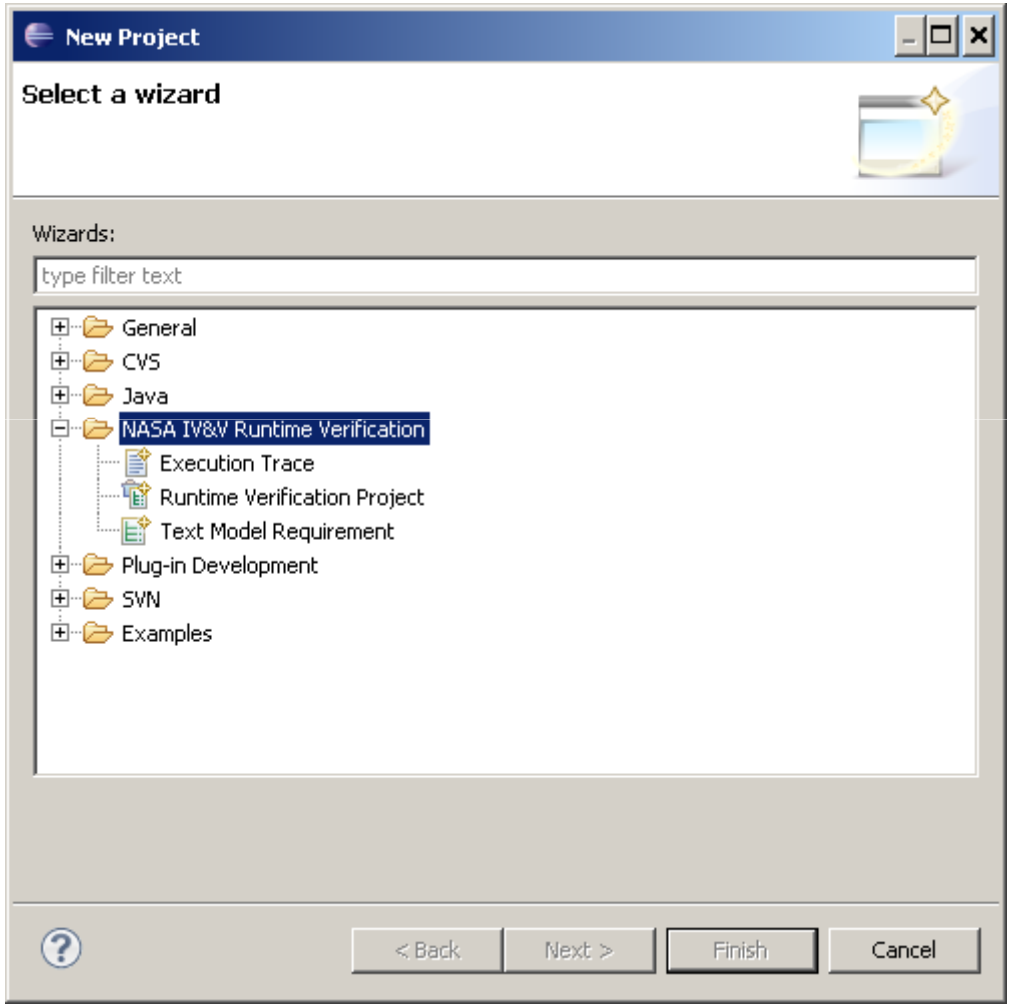


Eclipse Integration Overview

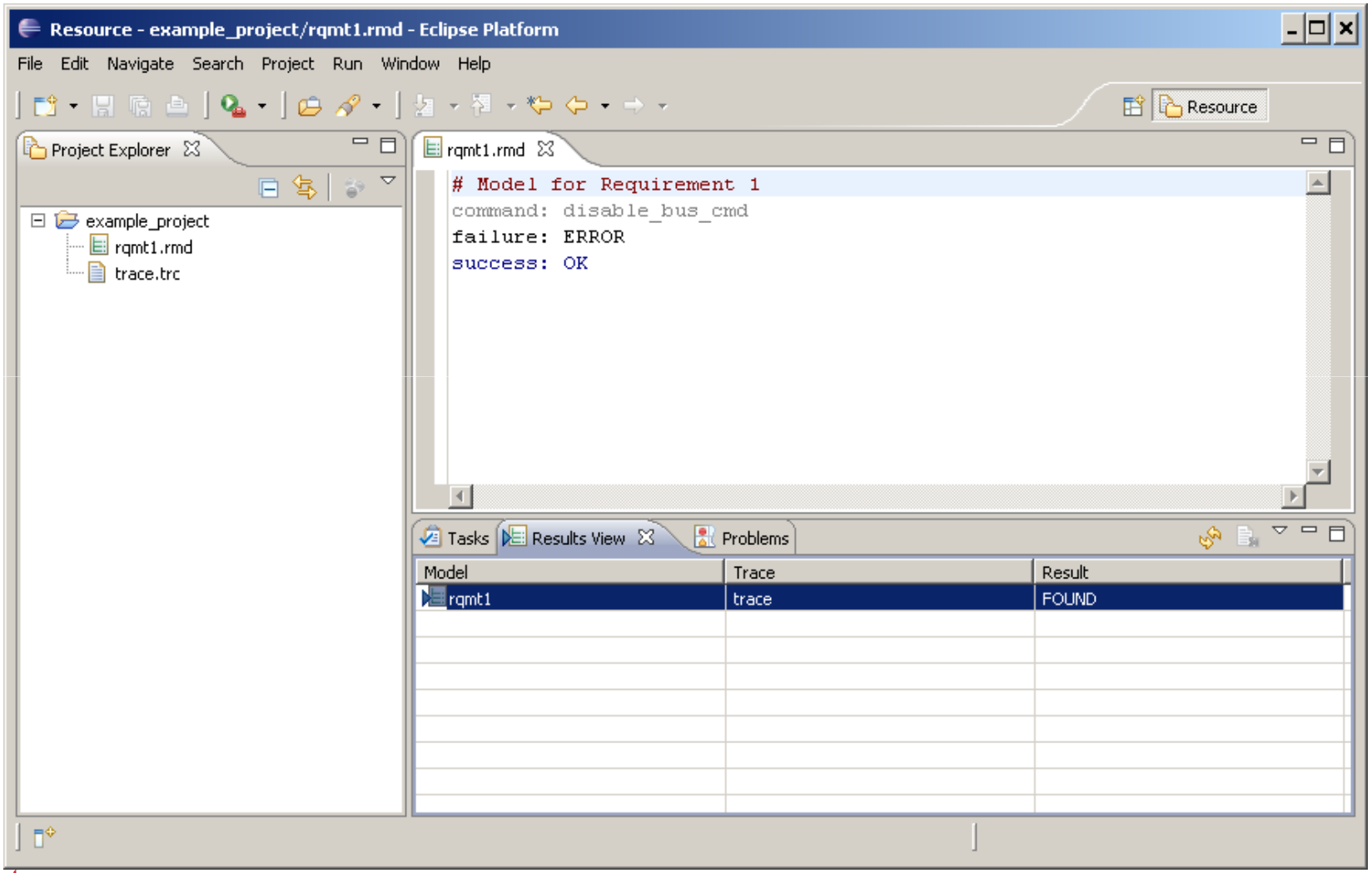
- Provides a new Eclipse project type called “NASA IV&V Runtime Verification.”
- Two new file types:
  - Text Model Requirement
  - Execution Trace
- Custom editors for both file types that includes syntax highlighting and error checking.

Eclipse Integration (1)

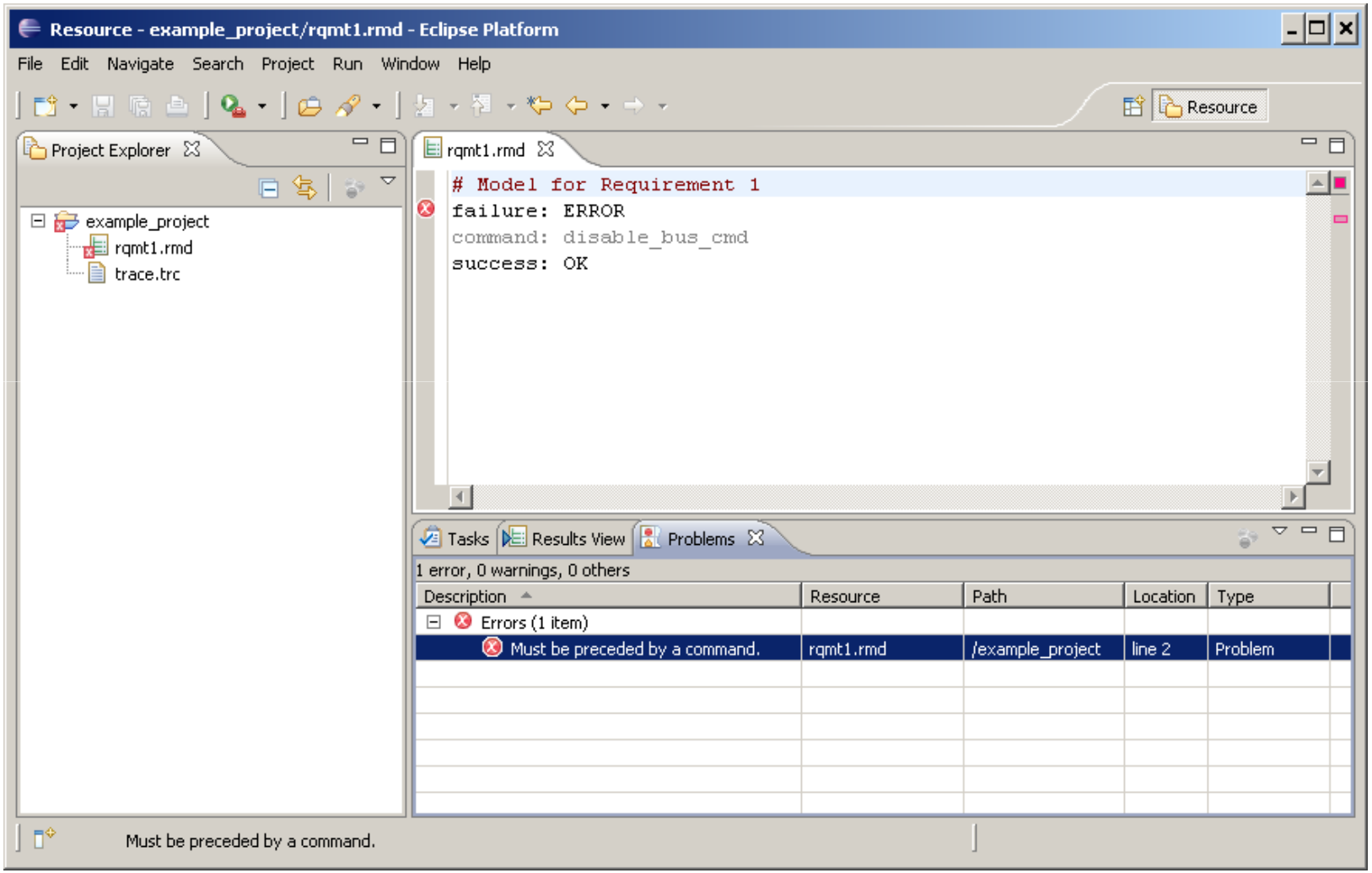
- Plug-ins allow for creating a “Runtime Verification” project.
- Project contains text models and execution traces.



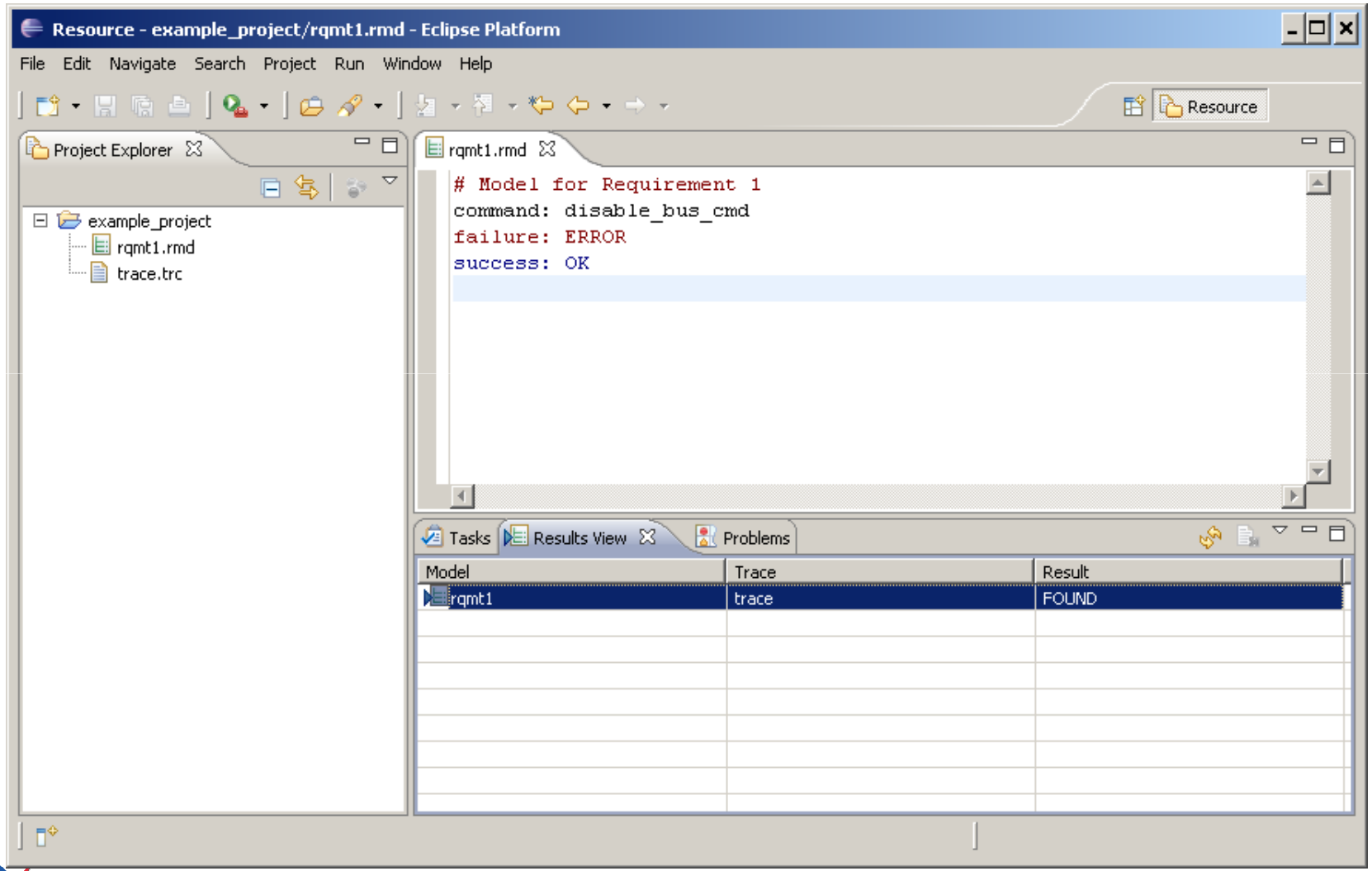
Eclipse Integration (2)



Eclipse Integration (3)



Eclipse Integration (4)



Model Checking

- Checks the execution trace for the model.
- Takes into consideration the:
  - Order of commands.
  - The depth of the call tree.
  - The distance between located commands.
- Will likely consider other factors as the algorithm development progresses.
- Will accommodate UML models once algorithm is sufficient.

Limitations

- Cannot test requirements that specify timing or latency constraints.
- Cannot test hardware-specific requirements without the flight hardware.

Summary

- Runtime Verification can provide:
  - Assurance that a requirement is implemented.
  - Confirmation of a non-implemented requirement.
  - Assertion checking to monitor states.
- Execution and profiling can provide:
  - Code coverage metrics:
    - Locate untested code.
    - Focus V&V efforts on code executed the most (80/20 rule).
  - Isolating requirements in unit tests provides the source code which implements that requirement.

Thank You

- Jeff Zemerick
- jeffrey.zemerick@tasc.com